LINKED LIST

Motivation

- Already used Array for storing list of items, which is a **Static memory allocation** process.

  int array[ ] = new int[5]

- Since array is static in nature, we can't add more elements beyond its capacity.

Limitations of Array

- ✓ Unused memory leads to memory loss
- ✓ Fixed Size
- ✓ Difficulty in inserting and deleting elements in array by means of shifting elements

Example

- Let’s consider that we need to store the names of all the students in a class.
- As number of students in the class is not given, we may store the names in an array with maximum size let 100.

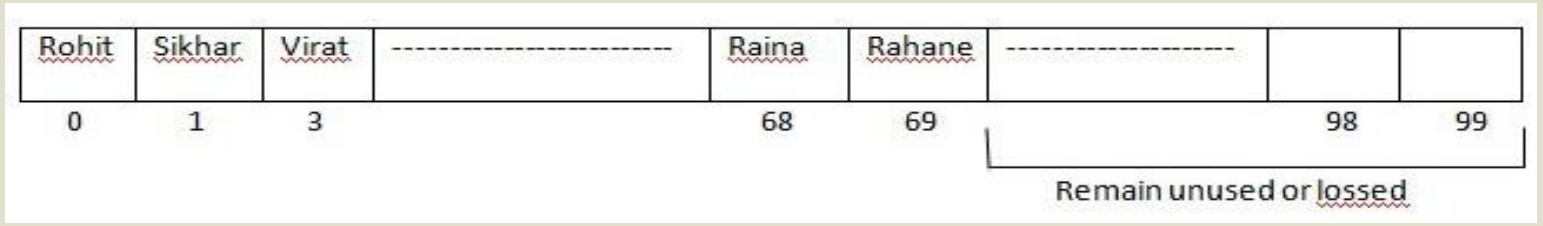


Fig 1. Array Representation

- Let the class contains only 70 students. As a result remaining memory locations are unused.
- If 103 students are there, then we can’t store 3 student’s information in the above array of size 100.
- Moreover, if we want to shuffle the names in the array, then it’ll be difficult to do so.

Better Solution

- In **linked list representation of data**, each name are stored in **separate non-contiguous memory blocks**, and each memory block is linked to next memory block.

- Linked list representation is **done dynamically** so that whenever we need/don't need extra memory block we can add/delete it.

- No need to predefine size of the linked list, so there is no memory loss, again we can add any number of memory blocks to the linked list because memory blocks are added to the linked list whenever required.

- Moreover insertion at any position is easier. We do not need any shifting operations , only we need to reset some fixed number of links. For example, if we need to insert name "Dhoni" after "Sikhar" then we need to reset only two links.

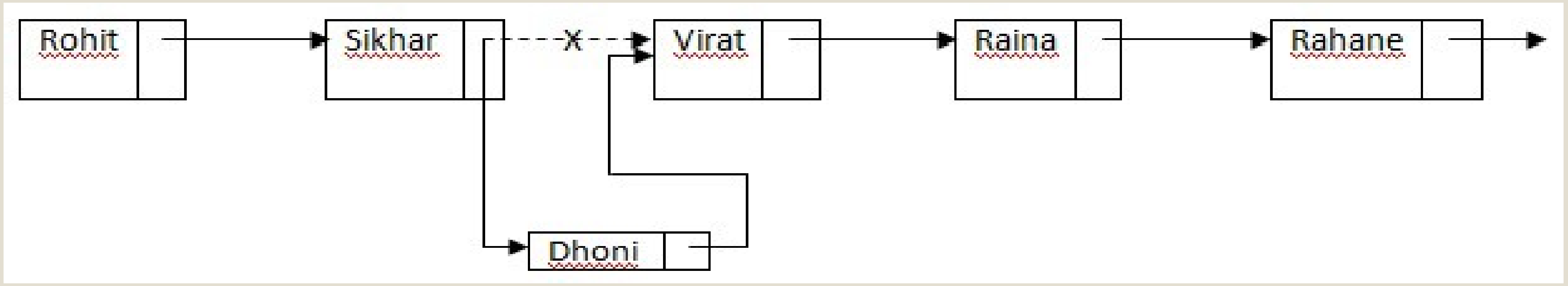


Fig 2. Linked List

Array vs Linked List

Array

- Random access of an element is possible because of the index values.
- Traversal is fast.
- Fixed size because memory is assigned during compile time.
- Difficulty in Insertion and deletion of elements.
- For e.g.- Phone contacts

Linked List

- Random access of an element is not possible because the list must begin from head/start.
- Traversal is slow.
- Dynamic size because memory is assigned during run time.
- Ease in insertion and deletion of elements.
- Use more memory because of the next pointer.
- Nodes are stored non-contiguously, greatly increasing the time to access elements.
- Difficulty in traversing backward.
- For e.g.- Music playlist

Types of Linked List

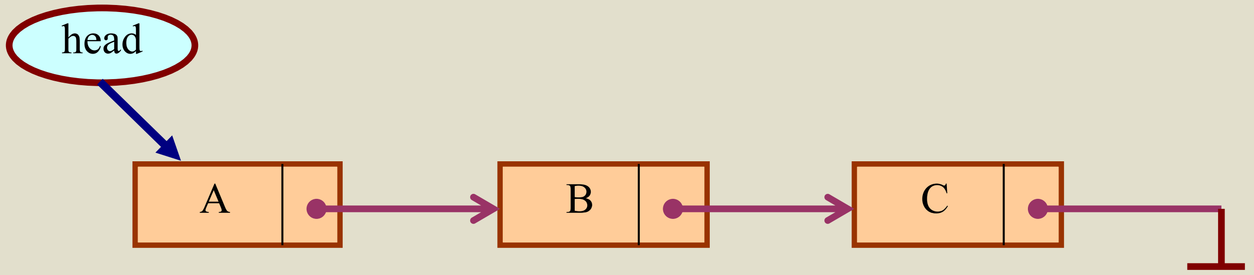


Fig 3. Singly Linked List

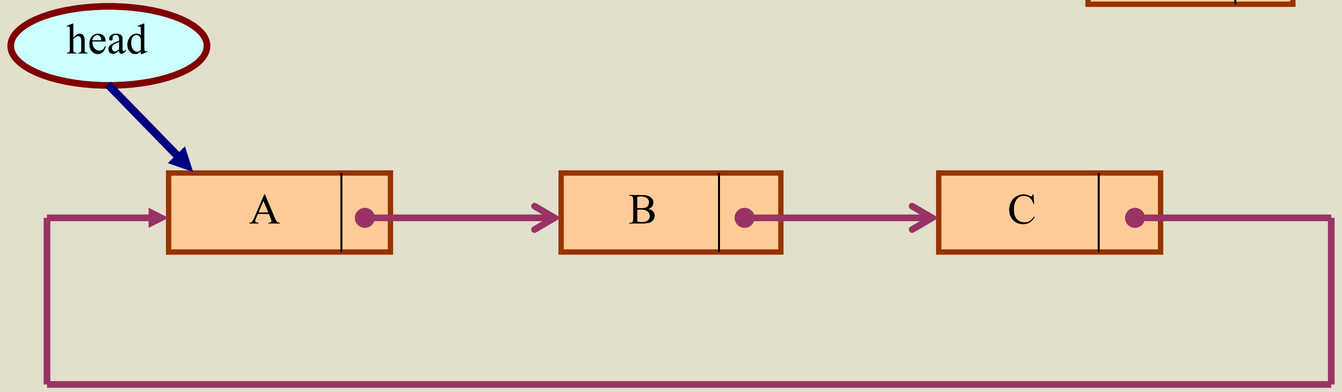


Fig 4. Circular Linked List

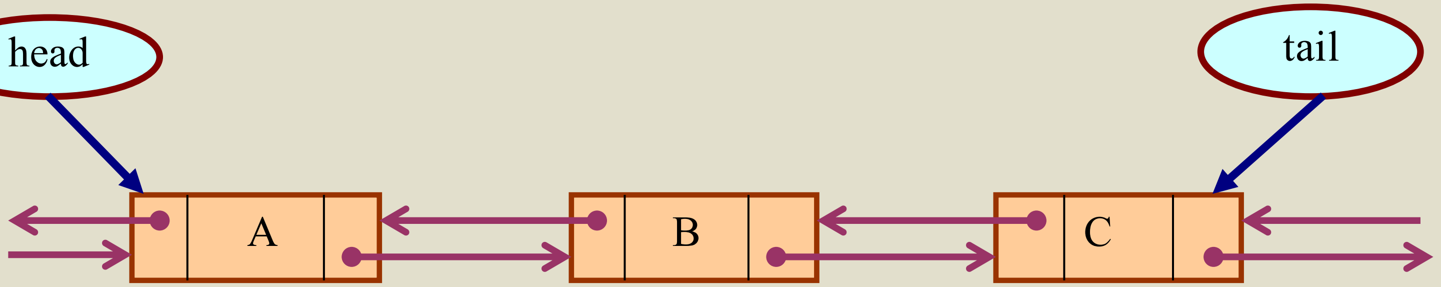


Fig 5. Doubly Linked List

Singly Linked List

- It is a collection of nodes and links.
- A node contains an Info and address of Next node.

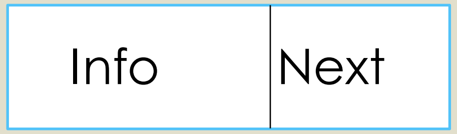


Fig 6. Node of a Single Linked List

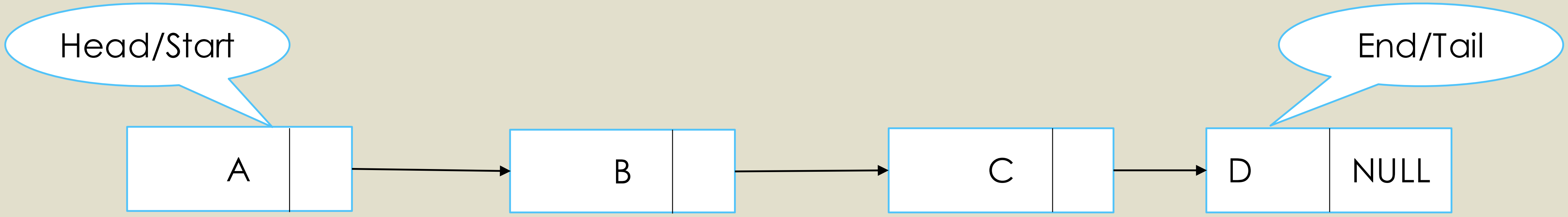


Fig 7. Example of a Linked List

- The last node of the list always contains NULL in its Next section.
- The first node always called as Head/Start and last node is End/Tail.

Operations in Linked List

- Create a node
- Insertion of new node
  - At beginning
  - At end
  - At any position
- Deletion of existing node
  - From beginning
  - From end
  - From any position
- Display/Traverse the list
- Count the number of nodes in the list
- Search an item in the list
- Sort the list
- Reverse the list

Creation

Initially, we must create a node (the first node) and make it as head.

```
class node
{
    char info;
    node next;
}

void create()
{
   Scanner sc=new Scanner (System.in);
   node head = null;                    // Initially linked list is empty
   node p = new node();                 // Allocate a memory block with address and name it as p.
   p.info = sc.next().charAt(0);        //Store data value in info part of p
   p.next = null;                       // set the link part to null
   head = p;                            // make p as head
}
```

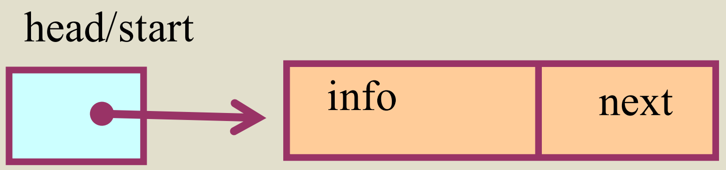


Fig 8. Node in Single Linked List

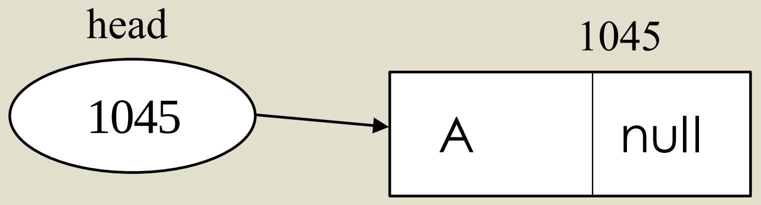
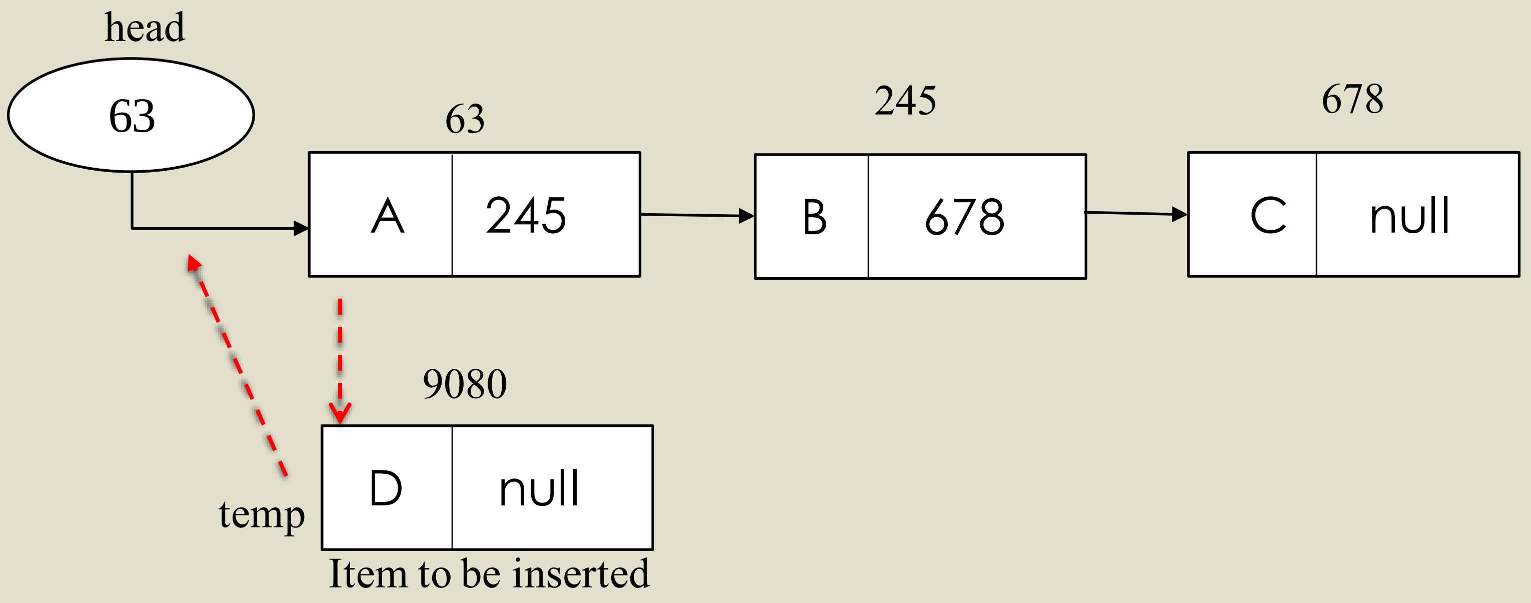


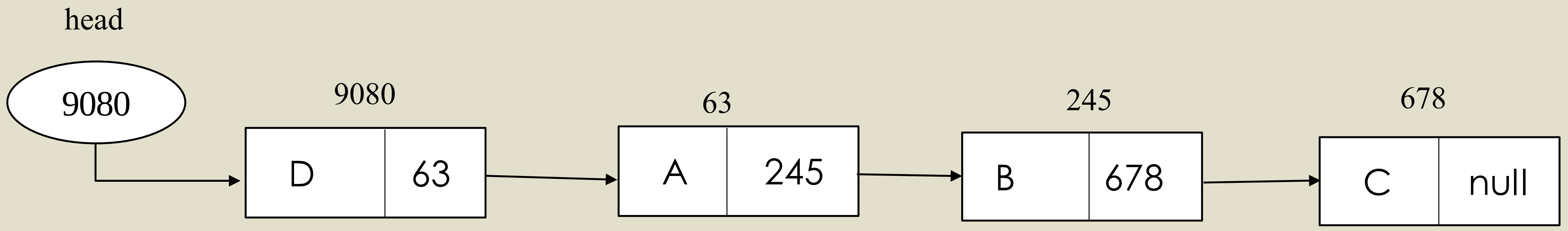
Fig 9. Example of a Node in Single Linked List

Insertion of new node

At beginning

```
void insert_beg()
{
    node temp = new node();
    temp.info = 'D';
    temp.next = null;
    temp.next = head;
    head = temp;
}
```

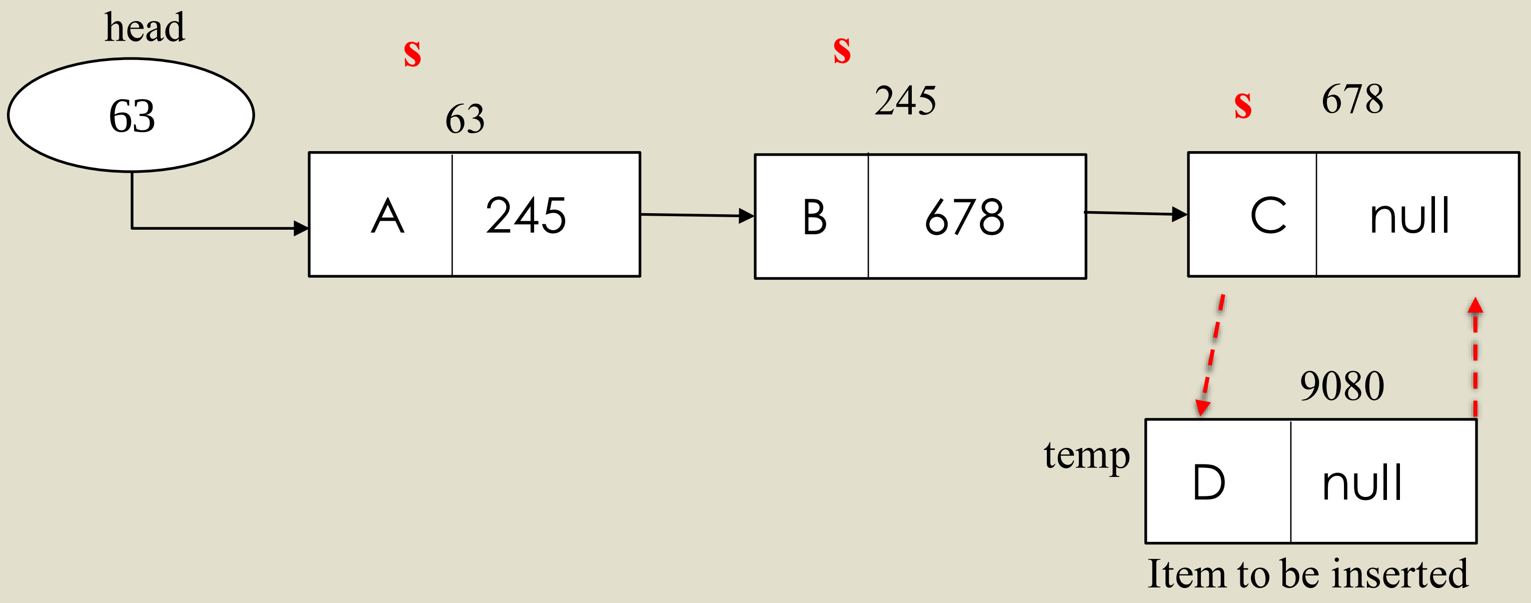


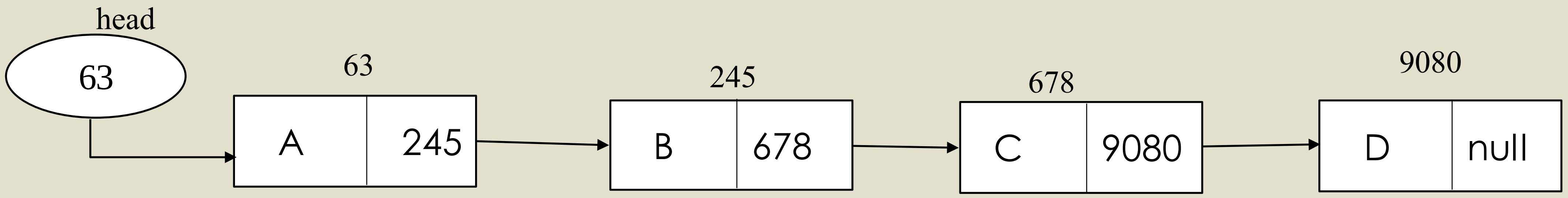


Insertion of new node

At end

```
void insert_end()
{
    node s = head;
    node temp = new node();
    temp.info = 'D';
    temp.next = null;
    while (s.next != null)
    {
        s = s.next;
    }
    s.next = temp;
}
```

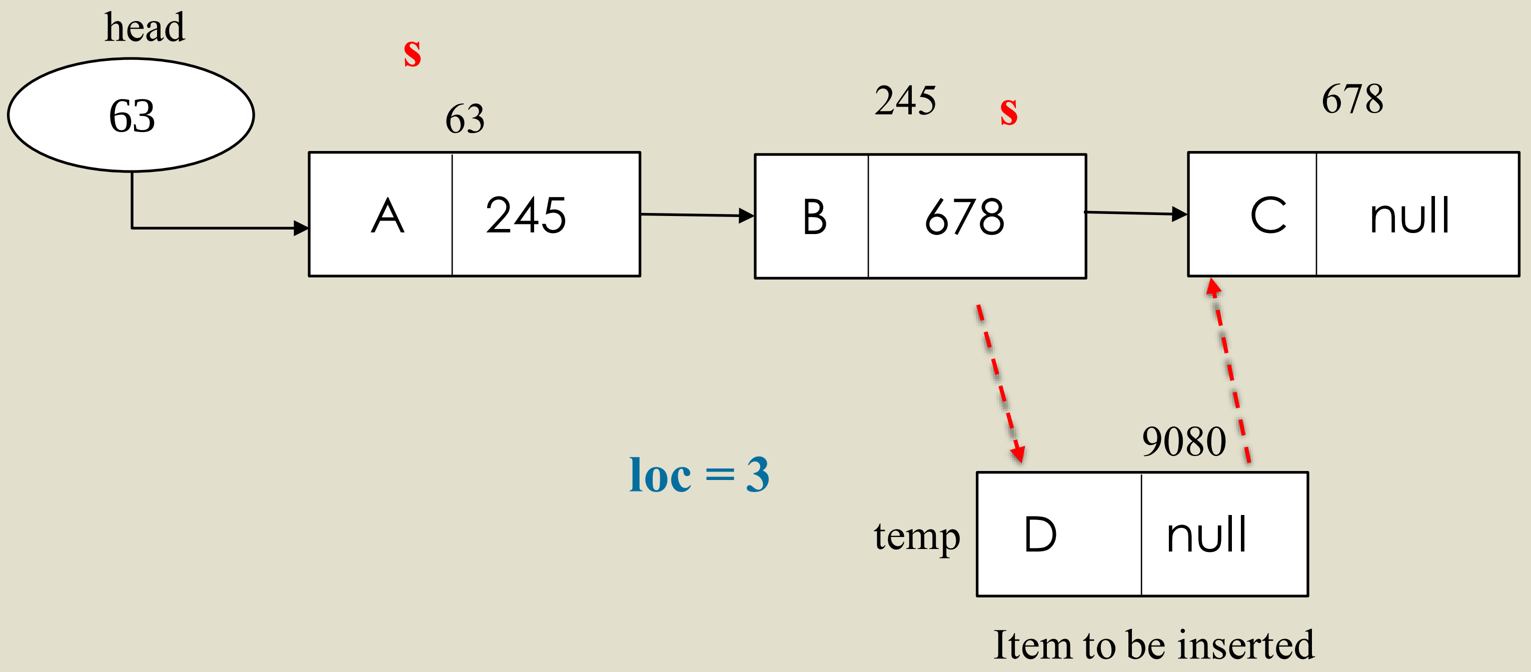


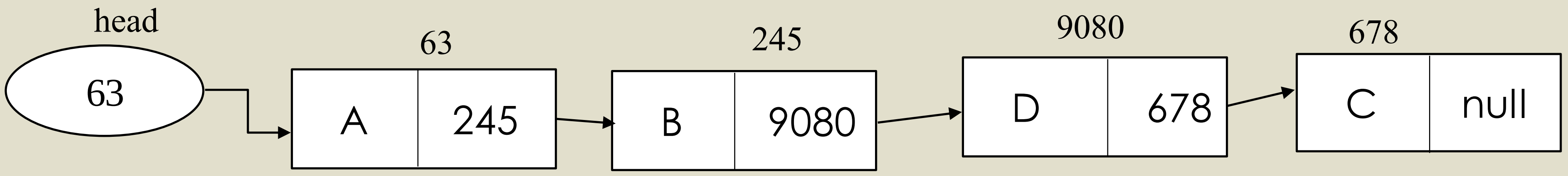


Insertion of new node

At any position

```
void insert_pos()
{
    node temp = new node();
    temp.info = 'D';
    temp.next = null;
    node s = head;
    System.out.println("Enter location.");
    int loc = sc.nextInt();
    for(int i=1; i < loc-1 && s.next != null; i++)
    {
        s = s.next;
    }
    temp.next = s.next;
    s.next = temp;
}
```

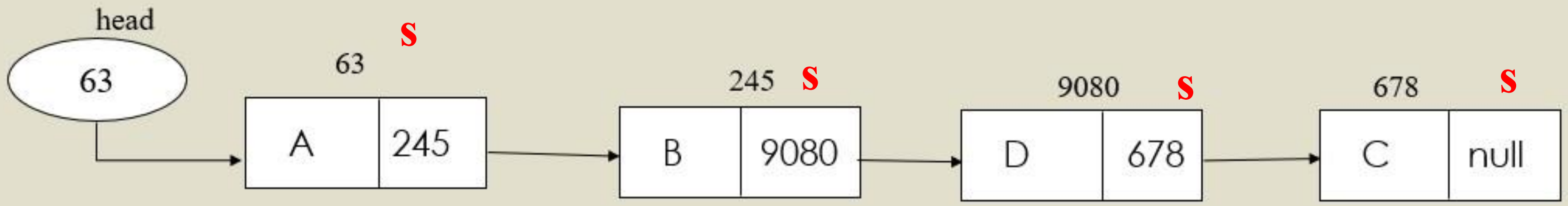




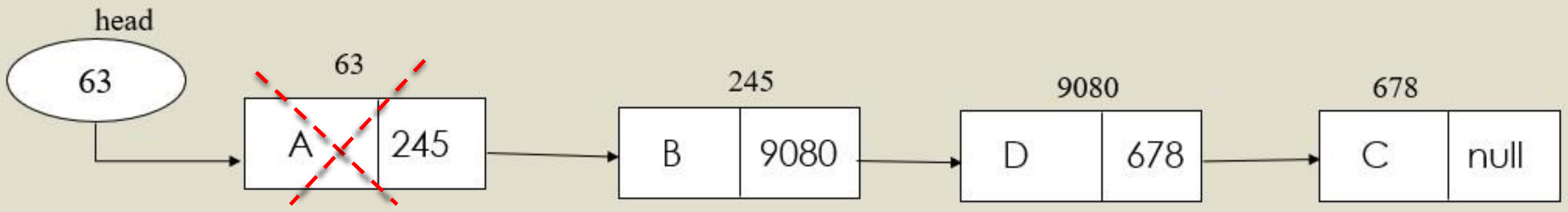
Display & Count

The list must begin from head.

```
void display()
{
   node s = head;
   int count = 0;
   while (s != null)
   {
         count ++;
         System.out.print (s.info + "   ->   ");
         s = s.next;
   }
    System.out.print (count);
}
```

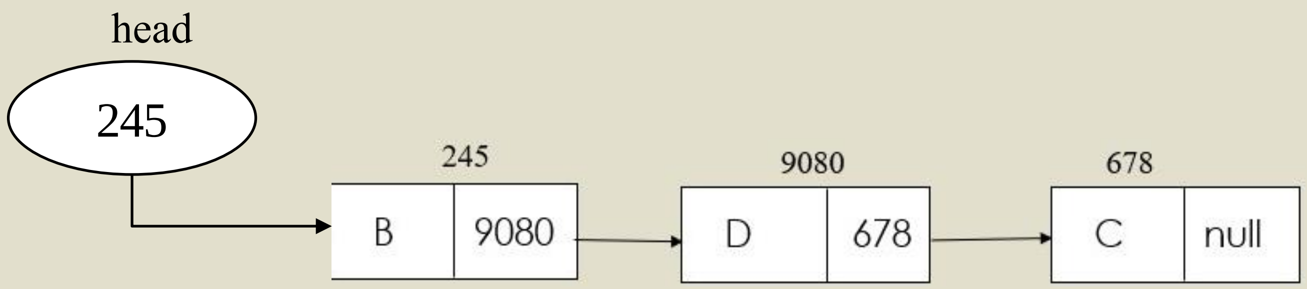


Deletion of a node



At beginning

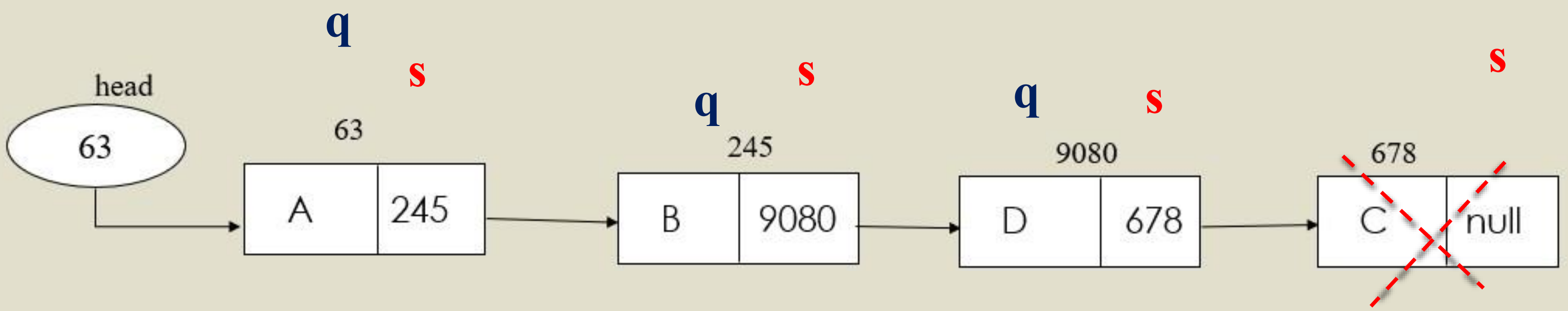
```
void delete_beg()
{
   System.out.println ("Node deleted at" + head);
   head = head.next;
}
```

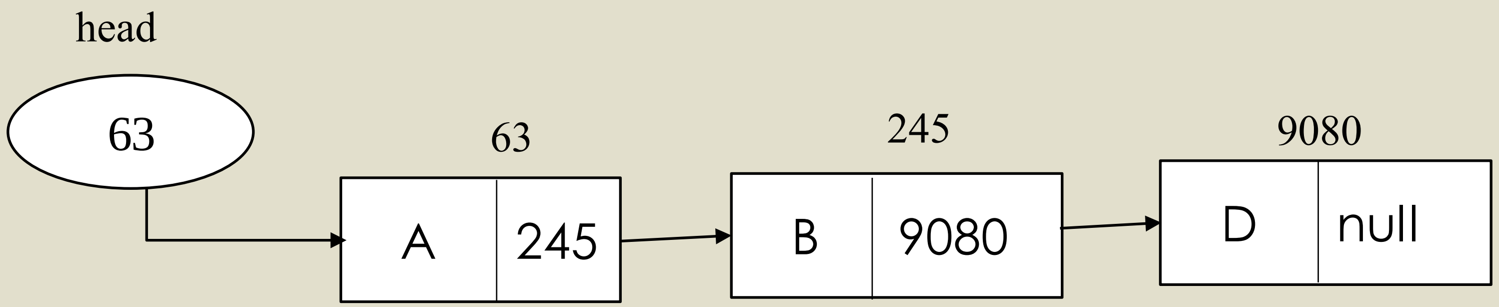


Deletion

At End

```
void delete_end()
{
   node s = head;
   node q = new node();
   while (s.next != null)
   {
      q = s;
      s = s.next;
   }
  System.out.println (“node deleted at: "+ s);
  q.next = null;
}
```

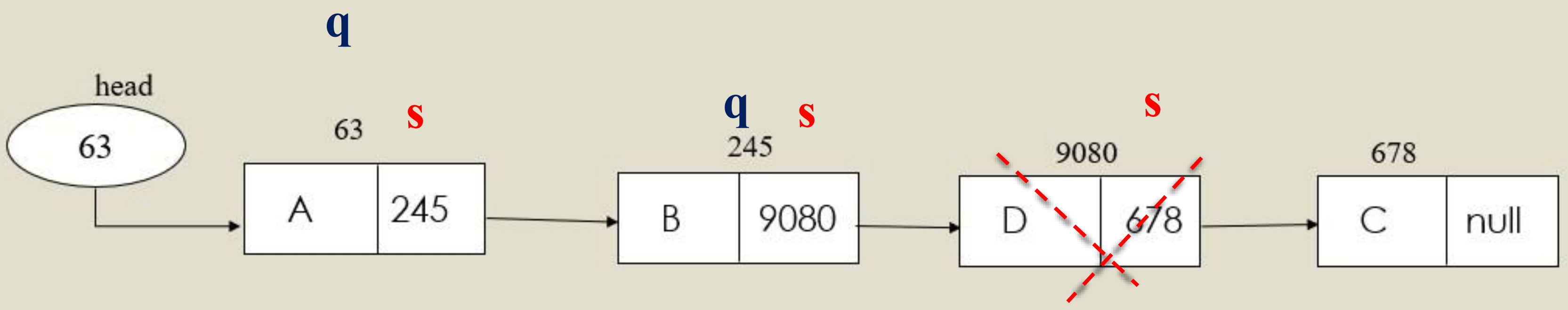


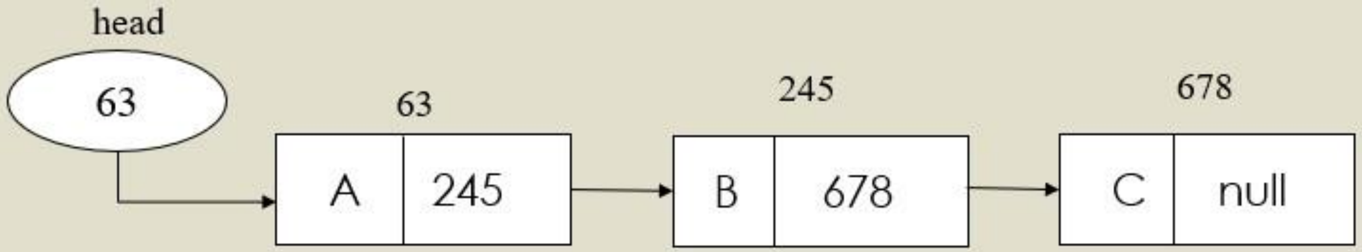


Deletion

At any position

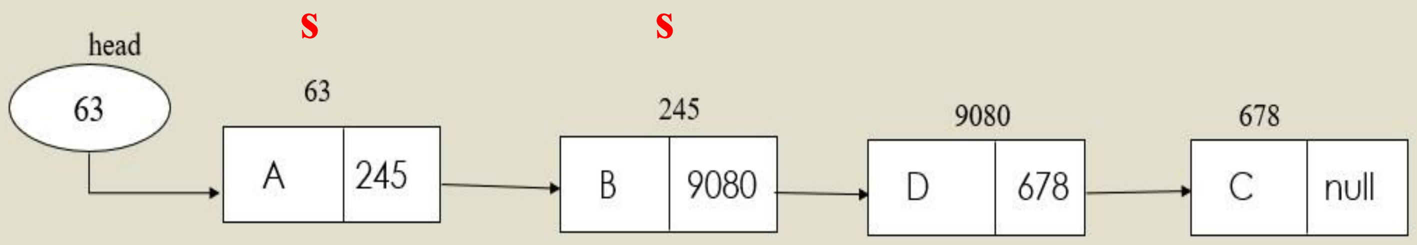
```
void delete_pos()
{
   node s = head;
   node q = new node();
   System.out.println("Enter location.");
    int loc = sc.nextInt();
    for(int i = 1; i < loc && s.next != null; i++)
    {
        q = s;
        s = s.next;
    }
    q.next = s.next;
}
```





Search an Item in the list

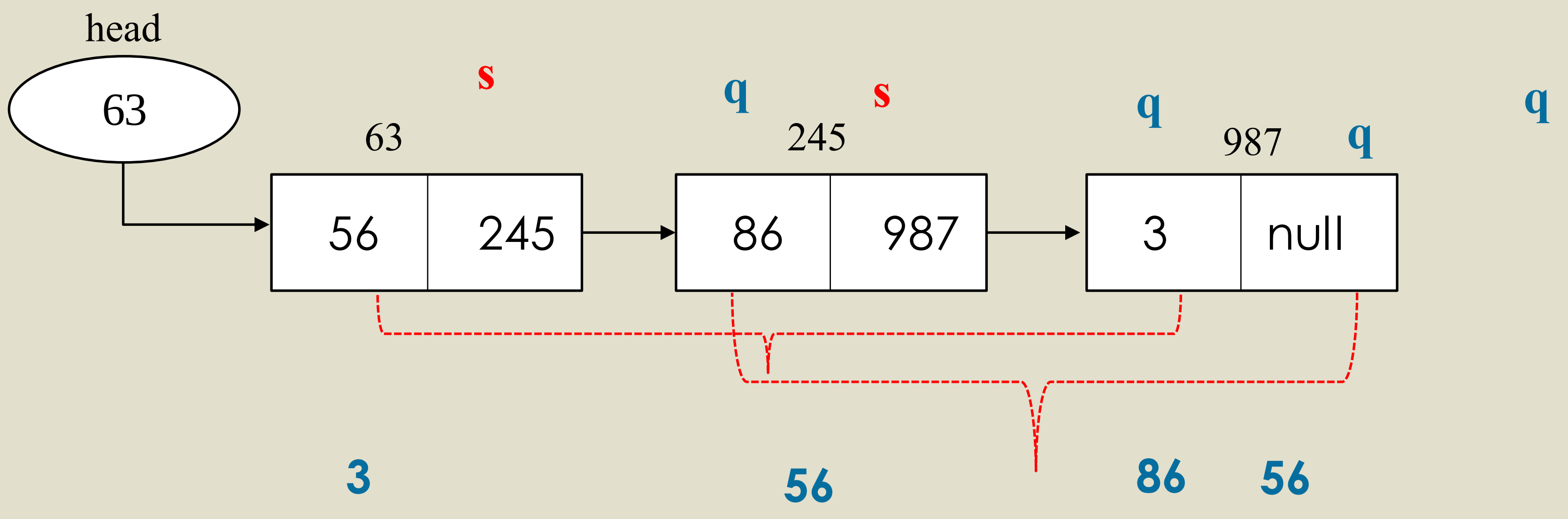
```
void search()
{
      node s = head;
      if (head = = null)
           System.out.println("List is empty.");
     else
     {
            Scanner sc = new Scanner (System.in);
            System.out.println ("Enter item to be searched");
            char x = sc.next().charAt(0);
            while (s != null)
            {
                  if ( x = = s.info)
                  {
                       System.out.print (“Item found");
                       break;
                  }
                  s = s.next;
            }
      }
}
```



x = B

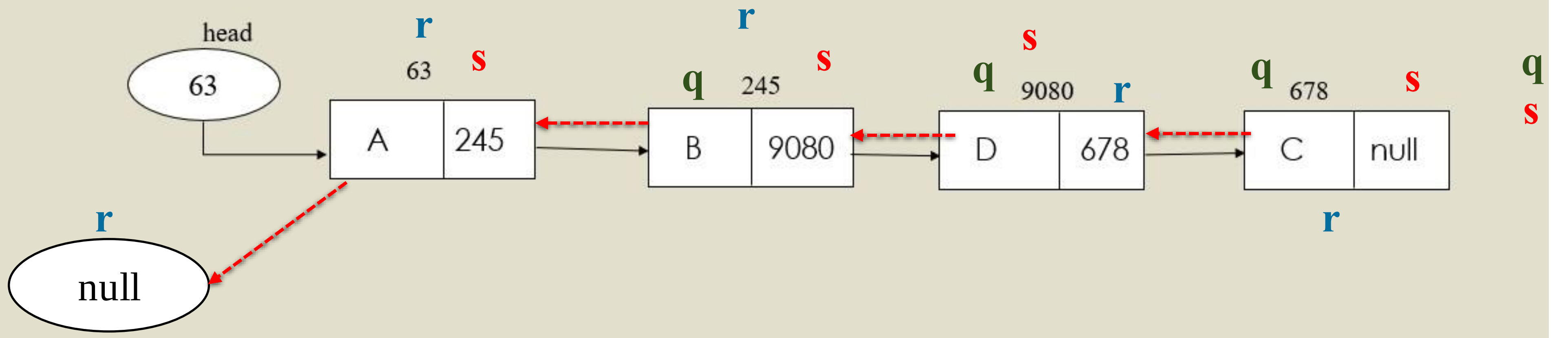
Sort a List

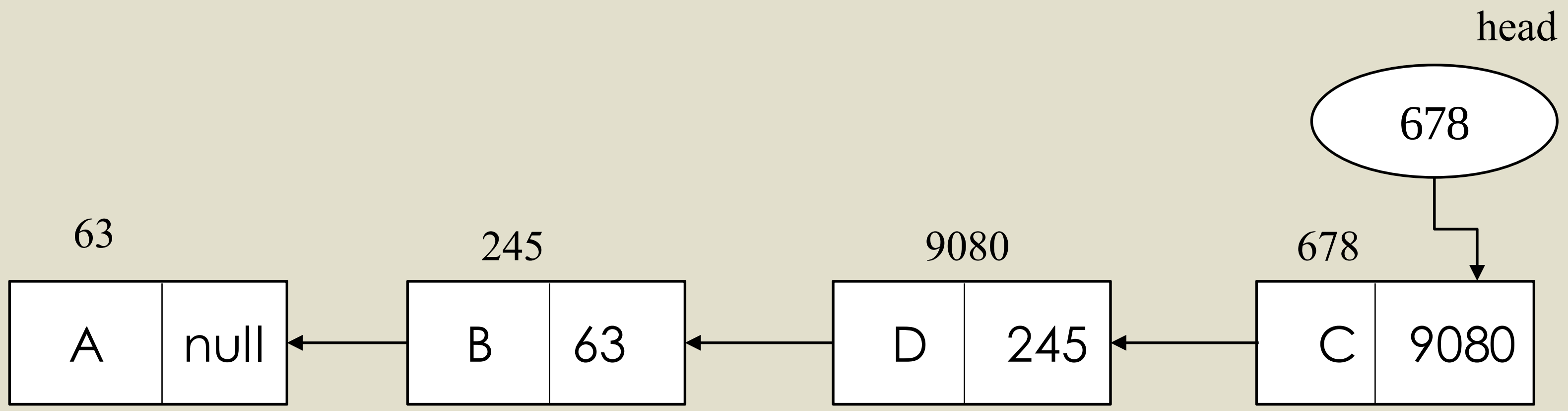
```
void sort()
{
    node s = head;
    node q = new node();
    while (s != null)
    {
      q = s.next;
      while (q != null)
      {
         if ( s.info > q.info)
         {
            int x = s.info;
            s.info = q.info;
            q.info = x;
         }
         q = q.next;
      }
     s = s.next;
   }
}
```



Reverse a List

```
void reverse()
{
   node s = new node();
   node r = new node();
   node q = new node();
   s = head;
   r = null;
   while (s != null)
   {
         q = s.next;
         s.next = r;
         r = s;
         s = q;
   }
   head = r;
}
```





Circular Linked List

- **Circular linked list** is a linked list where all nodes are connected to form a circle.
- There is no NULL at the end. Suppose the last node is denoted as **p**. Now, **p.next = head**.
- We can traverse the whole list by starting from any point.
- We just need to stop when the first visited node is visited again.

```
void display()
{
    node p = head;
    if (head = = null)
         System.out,println("List is empty");
    else
    {
        do
        {
            System.out.print (p.info + " ");
            p = p.next;
        }
        while (p != head);
    }
}
```

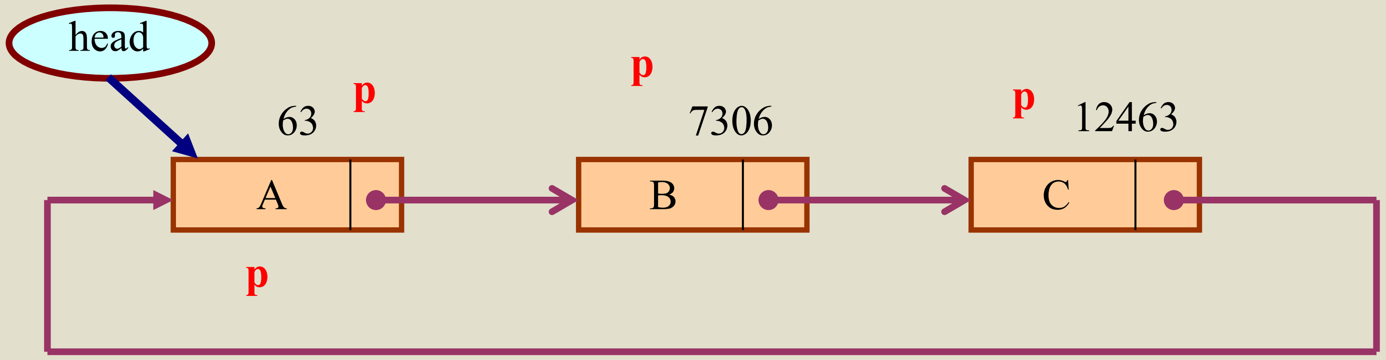


Fig 10. Circular Linked List

Doubly Linked List

- A node contains an Info and the address of Previous as well as the Next nodes.
- The last node of the list always contains NULL in its Next section.
- The first node always stores NULL in its Prev section.

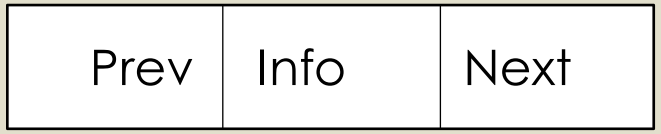


Fig 11. Node of a Double Linked List

```
class node
{
   char info;
   node prev, next;
}
```

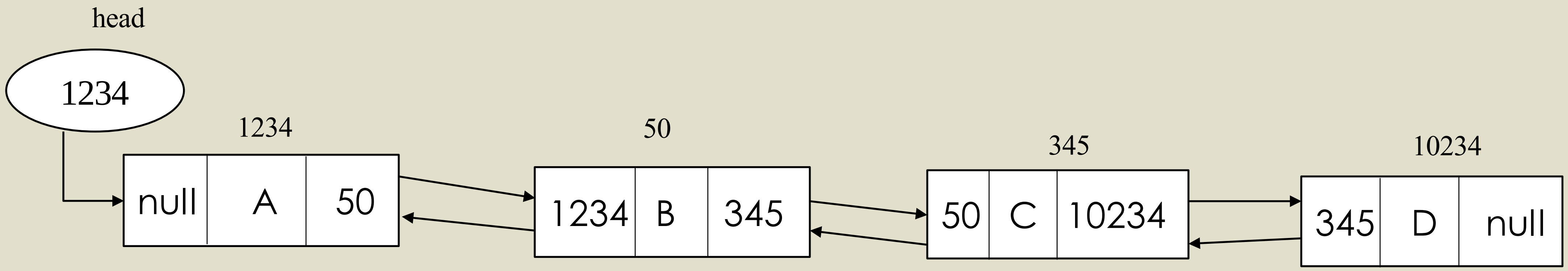


Fig 12. Example of a Doubly Linked List

Creation

Initially, we must create a node (the first node) and make it as head.

```
class node
{
    char info;
    node next, prev;
}

void create()
{
   Scanner sc=new Scanner (System.in);
   node head = null;                    // Initially linked list is empty
   node p = new node();                 // Allocate a memory block with address and name it as p.
   p.info = sc.next().charAt(0);        //Store data value in info part of p
   p.next = null;                       // set the next part to null
   p.prev = null;                       // set the prev part to null
   head = p;                            // make p as head
}
```

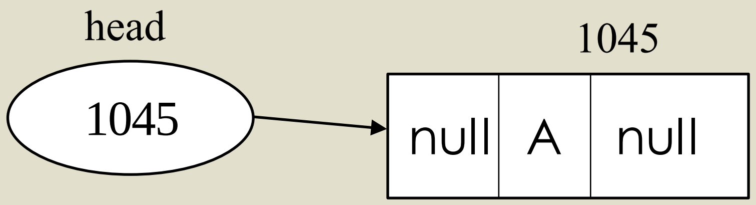
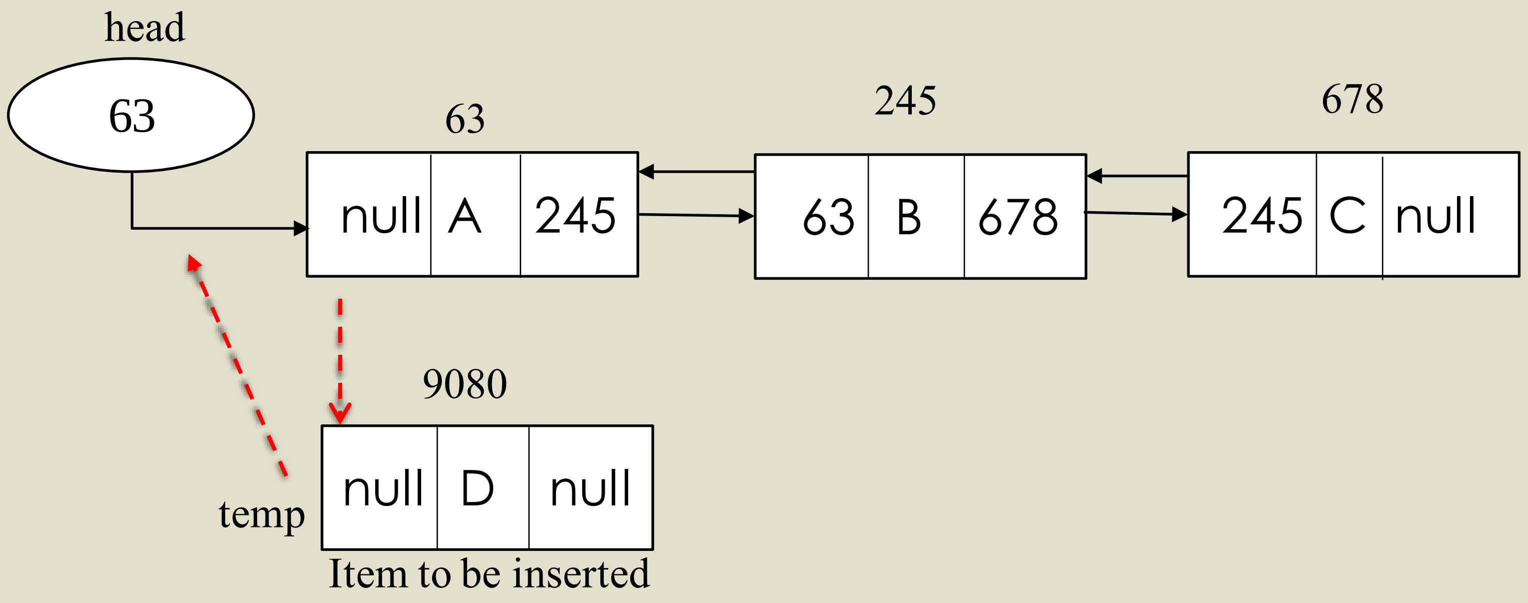


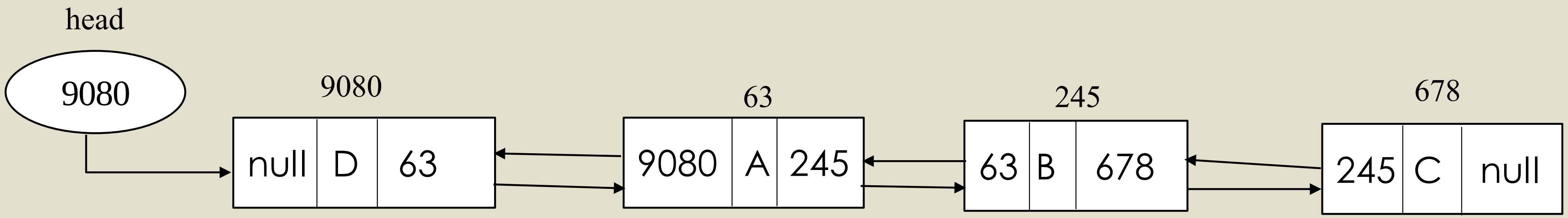
Fig 13. Node in Doubly Linked List

Insertion of new node

At beginning

```
void insert_beg()
{
    node temp = new node();
    temp.info = 'D';
    temp.prev = null;
    temp.next = head;
    head.prev = temp;
    head = temp;
}
```

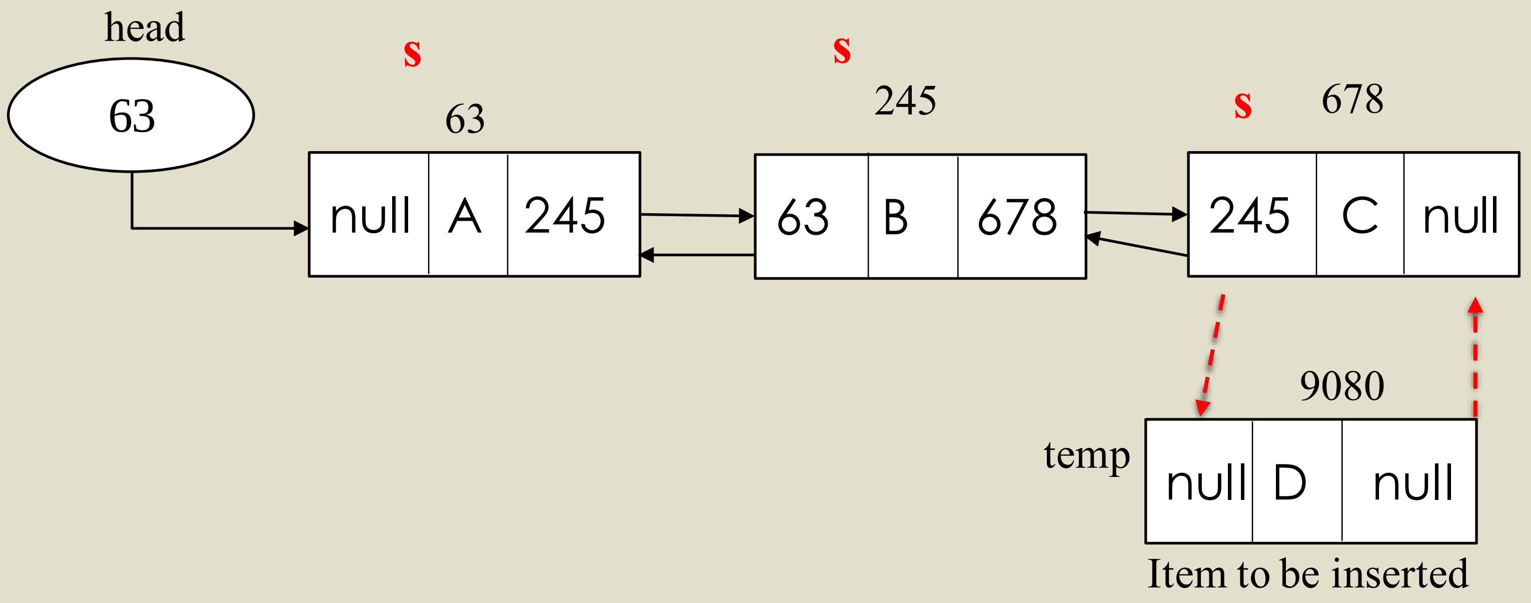


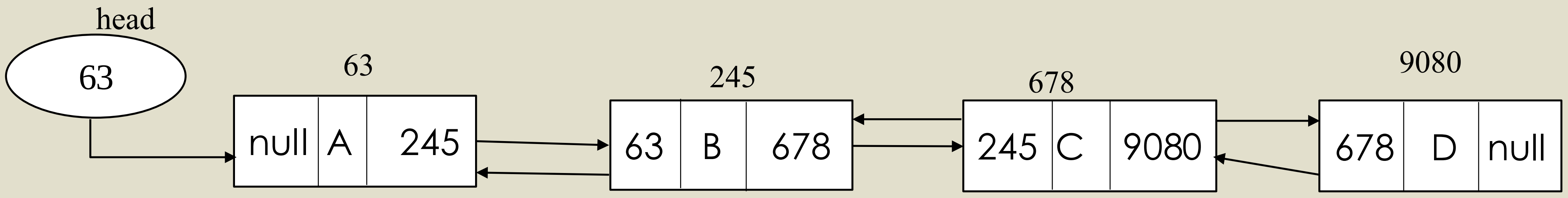


Insertion of new node

At end

```
void insert_end()
{
    node s = head;
    node temp = new node();
    temp.info = ‘D’;
    temp.next = null;
    while (s.next != null)
    {
        s = s.next;
    }
    s.next = temp;
    temp.prev = s;
}
```

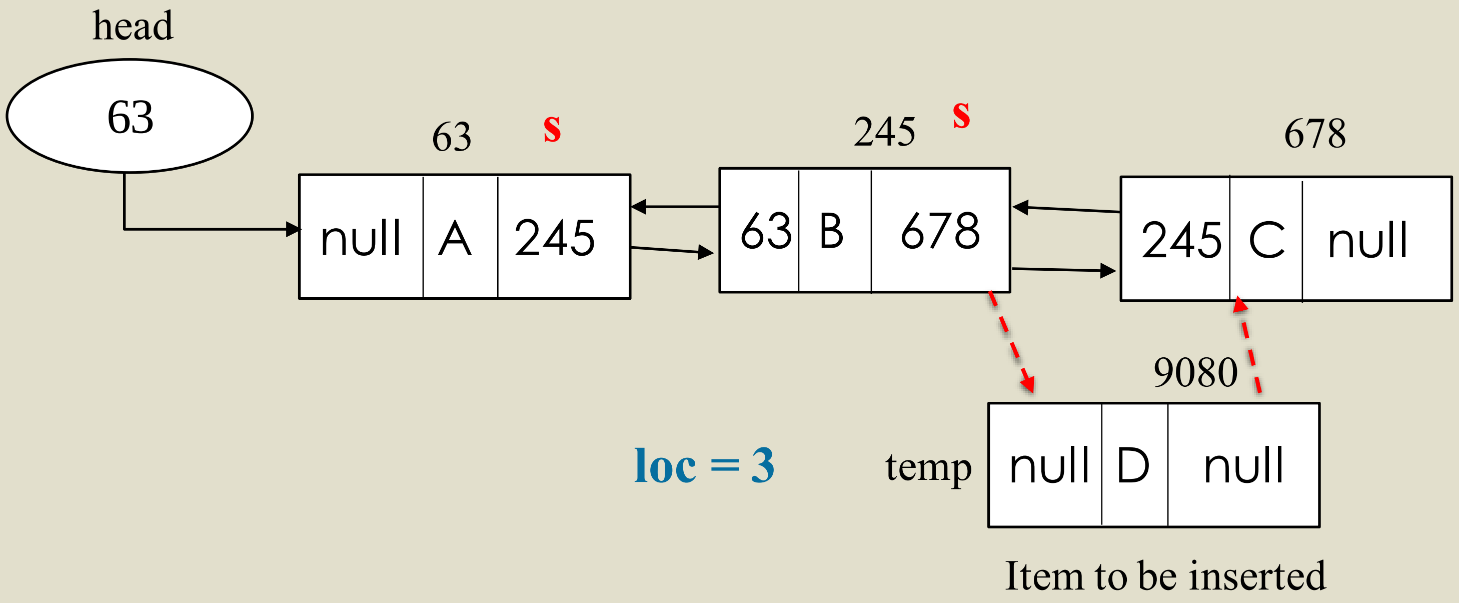


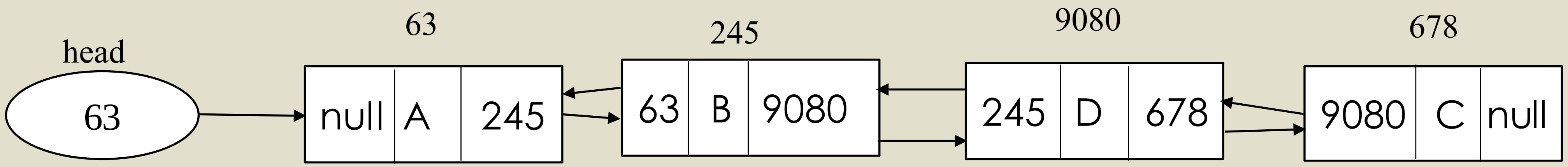


At any position

Insertion of new node

```
void insert_pos()
{
    node temp = new node();
    temp.info = 'D';
    temp.next = null;
    node s = head;
    System.out.println("Enter location.");
    int loc = sc.nextInt();
    for(int i=1; i < loc-1 && s.next != null; i++)
   {
        s = s.next;
   }
    temp.prev = s;
    temp.next = s.next;
    s.next.prev = temp;
    s.next = temp;
}
```





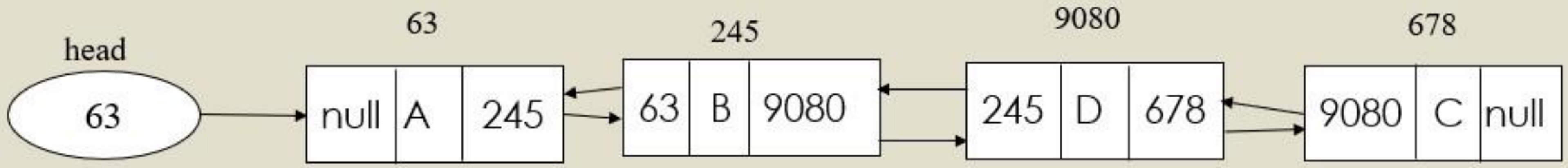
Display

Forward display

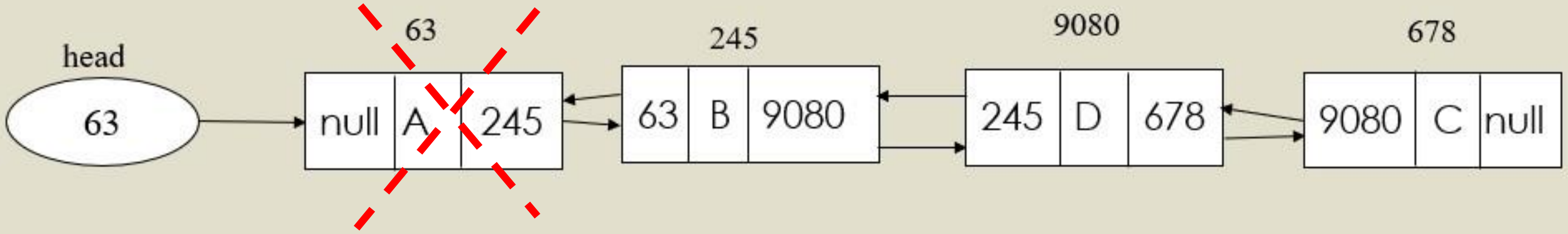
```
void display()
{
    node s = head;
    while (s != null)
   {
          System.out.print (s.info + "   ->   ");
          s = s.next;
    }
}
```

Backward display

```
void display()
{
    node s = head;
    while (s.next != null)
     {
          s = s.next;
      }
    while (s != null)
    {
          System.out.print (s.info + "   ->   ");
          s = s.prev;
    }
}
```

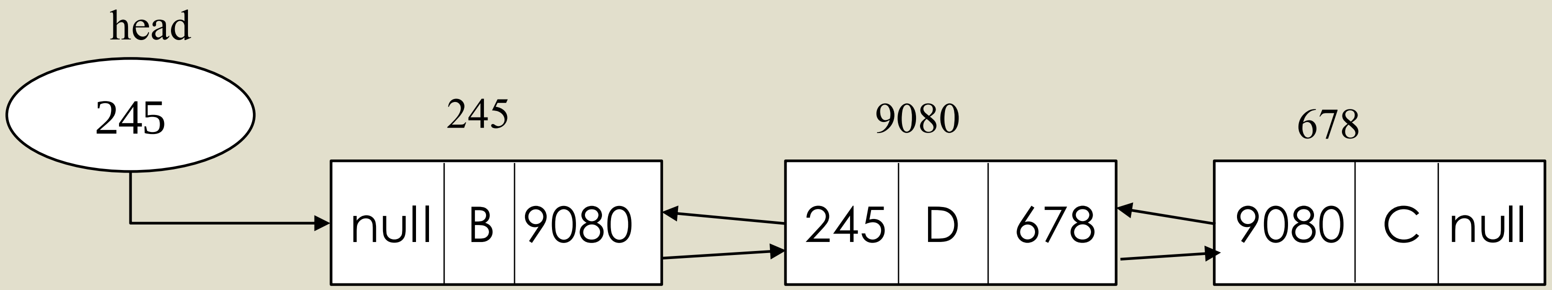


Deletion of a node



At beginning

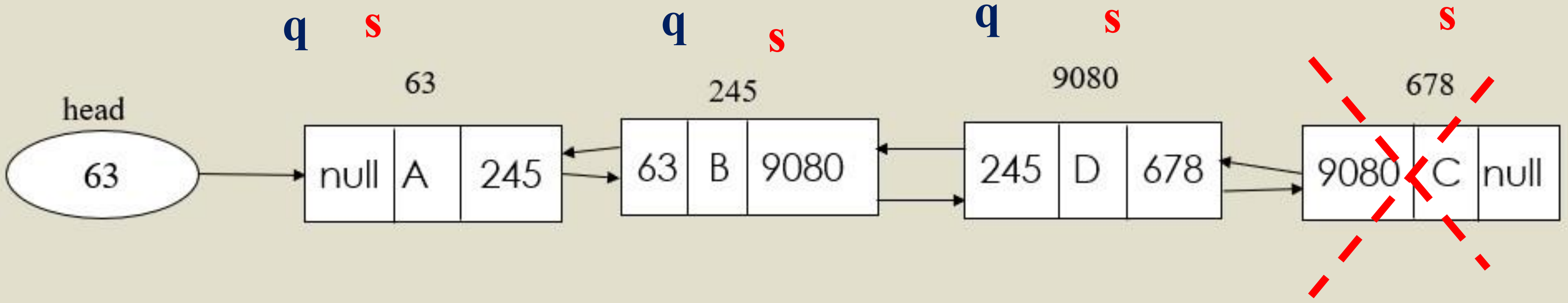
```
void delete_beg()
{
   System.out.println ("Node deleted at" + head);
   head = head.next;
   head.prev = null;
}
```

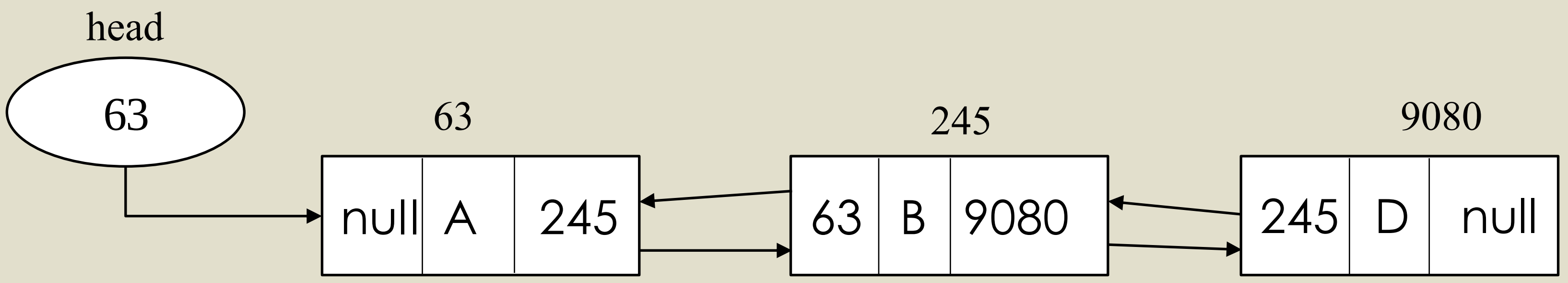


Deletion

At End

```
void delete_end()
{
   node s = head;
   node q = new node();
   while (s.next != null)
   {
      q = s;
      s = s.next;
   }
  System.out.println (“node deleted at: "+ s);
  q.next = null;
}
```

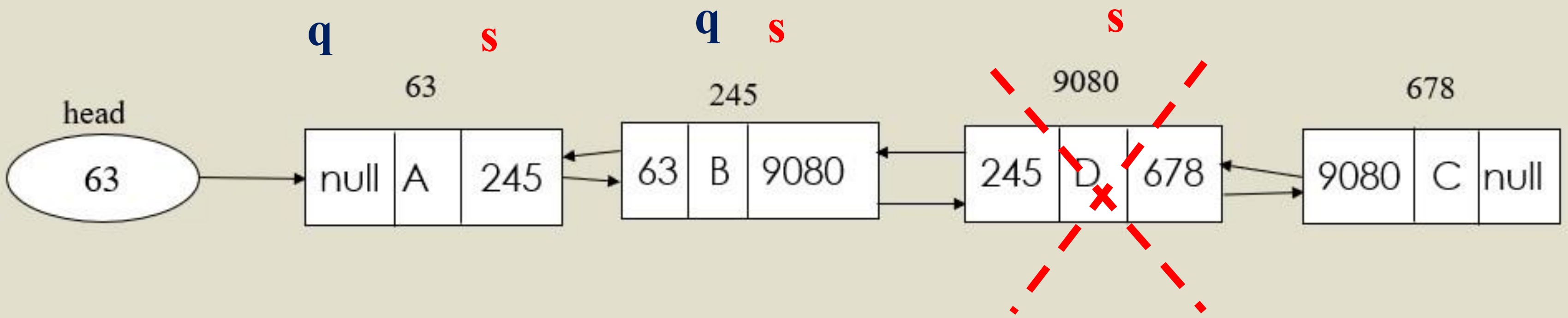


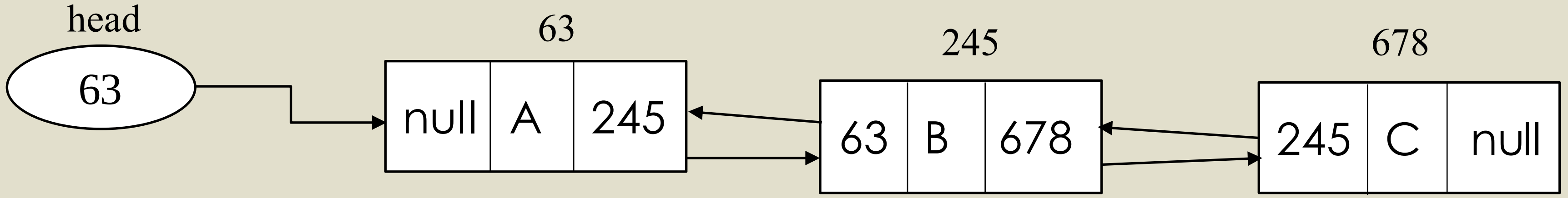


Deletion

At any position

```
void delete_pos()
{
   node s = head;
   node q = new node();
   System.out.println("Enter location.");
    int loc = sc.nextInt();
    for(int i = 1; i < loc && s.next != null; i++)
    {
        q = s;
        s = s.next;
     }
     q.next = s.next;
     s.next.prev = q;
}
```


q
s
q
s
s
head
63
63
null
A
245
245
63
B
9080
9080
245
D
678
678
9080
C
null


head
63
63
null
A
245
245
63
B
678
678
245
C
null